Storage Formats

CELEBRATING 30 YEARS
Marlan and Rosemary Bourns
College of Engineering

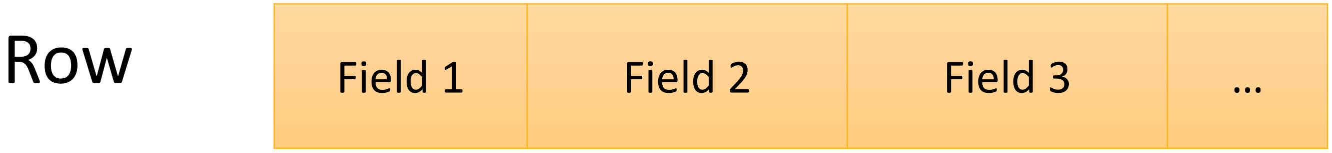
Overview

- We covered storage of unstructured files in HDFS
  - Partition into blocks
  - Replicate to data nodes
  - HDFS treats each file as a sequence of data, i.e., it is data agnostic
- This lecture covers an HDFS-friendly format for nested semi-structured data

Data Normalization

- In RDBMS, data must be at least in 1-NF
  - Think of it as a spreadsheet
  - Each row represents a record
  - Each column represents a field
  - You can have only one primitive value for each cell, possibly null
- In the big-data world, data is not in 1-NF
  - JSON is the standard format
  - JSON allows nesting and repetition (lists)
  - How to efficiently store this data in HDFS?

Row-oriented Stores

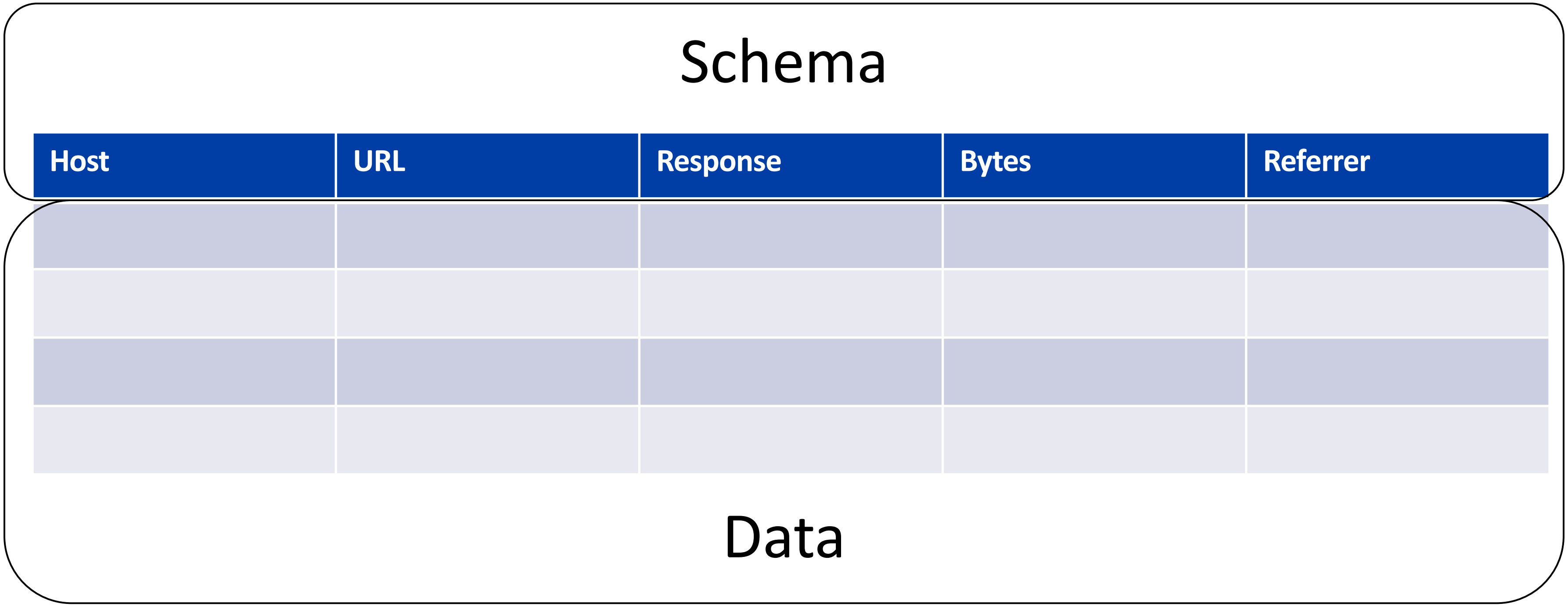


Row

Field 1	Field 2	Field 3	…

- CSV and JSON formats are examples of traditional row-oriented data formats
- CSV is naturally in 1-NF, similar to spreadsheets
- JSON supports nesting and repetition
- Q: How is the schema defined for in CSV and JSON?

CSV Schema Definition



Schema

Host	URL	Response	Bytes	Referrer

Data

- Advantage: Low overhead
- Disadvantages: Rigid model (not flexible), does not support nesting

JSON Schema Definition

```
{
  “created-at”: “Mon May 06 20:01:29 +0000 2019”,
  “id”: 9457298472,
  “text”: “Good Morning!”,
  “user”: {
    “id”: 242342,
    “name”: “Alex”,
    “location: {“city”: “Riverside”, “state”, “CA”,
“country”: “USA”}
  }
```

- Advantages: Flexible model. Supports nesting.
- Disadvantage: High overhead. Schema is repeated for each record

Row Format

- Both CSV and JSON are considered row formats when stored in their textual form
- Row formats is beneficial when the entire record needs to be processed as one unit
- Traditional RDBMS use row formats
- How about analytical queries?
  - Count of records
  - Sum of bytes
  - Avg(bytes) per response code

Column Format

- Stores each column separately rather than each row

ID	Name	Email	…
1	Jack	jack@example.com	
2	Jill	jill@example.net	
3	Alex	alex@example.org	

ID
1
2
3

Name
Jack
Jill
Alex

Email
…
…
…

Column Format

- Preferred for analytical queries that access a few set of columns, e.g., avg(bytes) per response code
- Can avoid reading unnecessary attributes from disk
- Columns can be encoded more efficiently
  - Bit masks for null value
  - Delta encoding
  - Run-length encoding (RLE)
- Column format is preferred in data warehouses

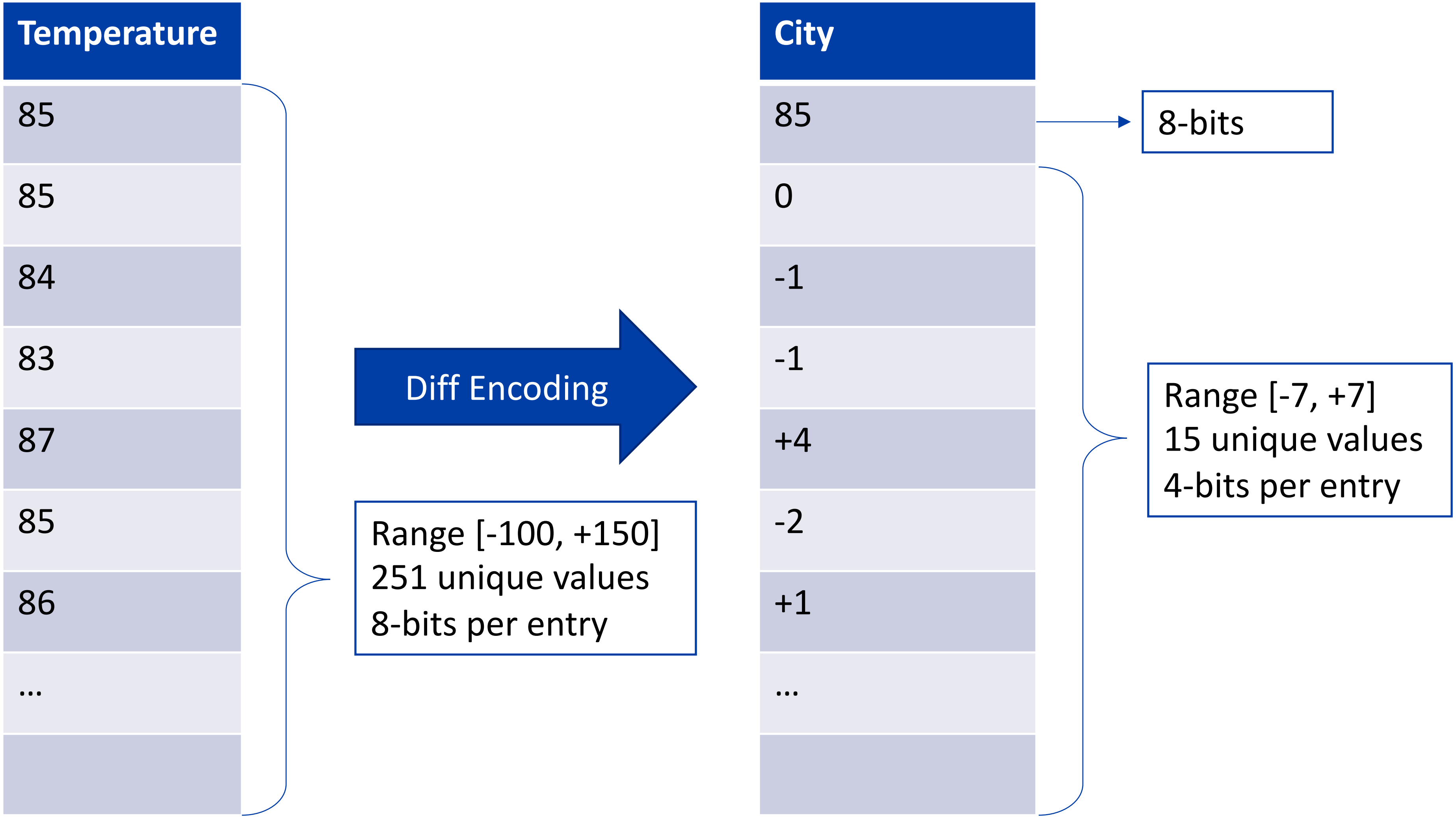
Column Encoding/Compression

City
Riverside
Riverside
Riverside
Los Angeles
Los Angeles
Riverside
Sacramento
…

Run Length Encoding (RLE)

City
Riverside,3
Los Angeles,2
Riverside, 1
Sacramento, 1
…

Column Encoding/Compression



Temperature
85
85
84
83
87
85
86
…

Diff Encoding

Range [-100, +150]
251 unique values
8-bits per entry

City
85
0
-1
-1
+4
-2
+1
…

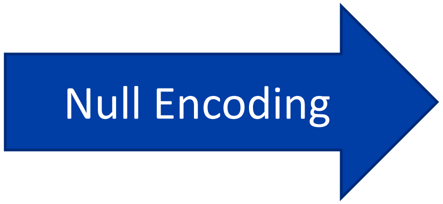
8-bits

Range [-7, +7]
15 unique values
4-bits per entry

Compression/decompression cost
One (or a few) instruction per value

Encoding of Null Values

Email
jeff@example.com
chen@example.org
alex@example.net
nora@example.com
…

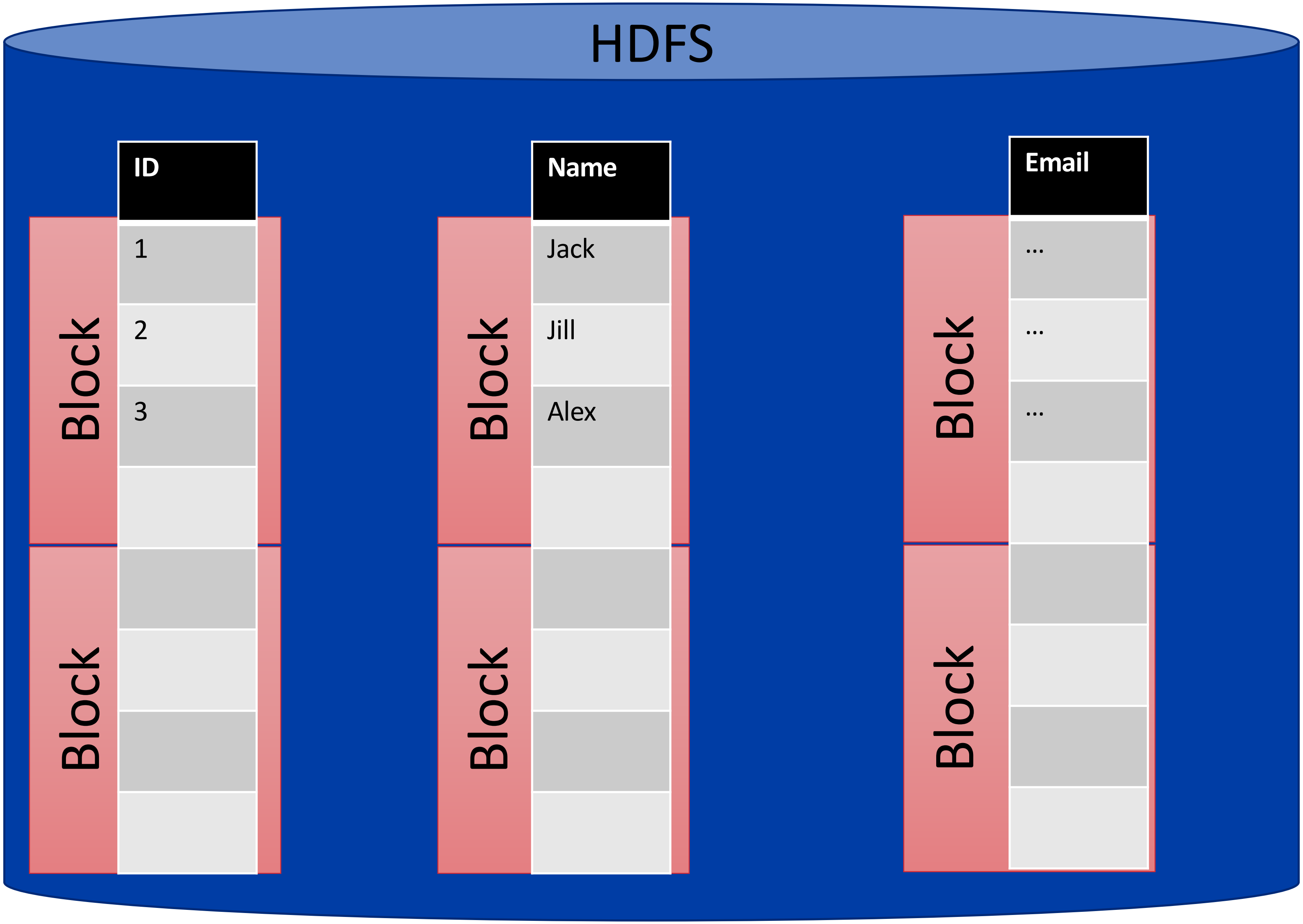


Null Encoding

Exists
1
0
1
0
0
0
1
0
1
…

Email
jeff@example.com
chen@example.org
alex@example.net
nora@example.com
…

Column Format for Big Data



ID	Name	Email
1	Jack	…
2	Jill	…
3	Alex	…

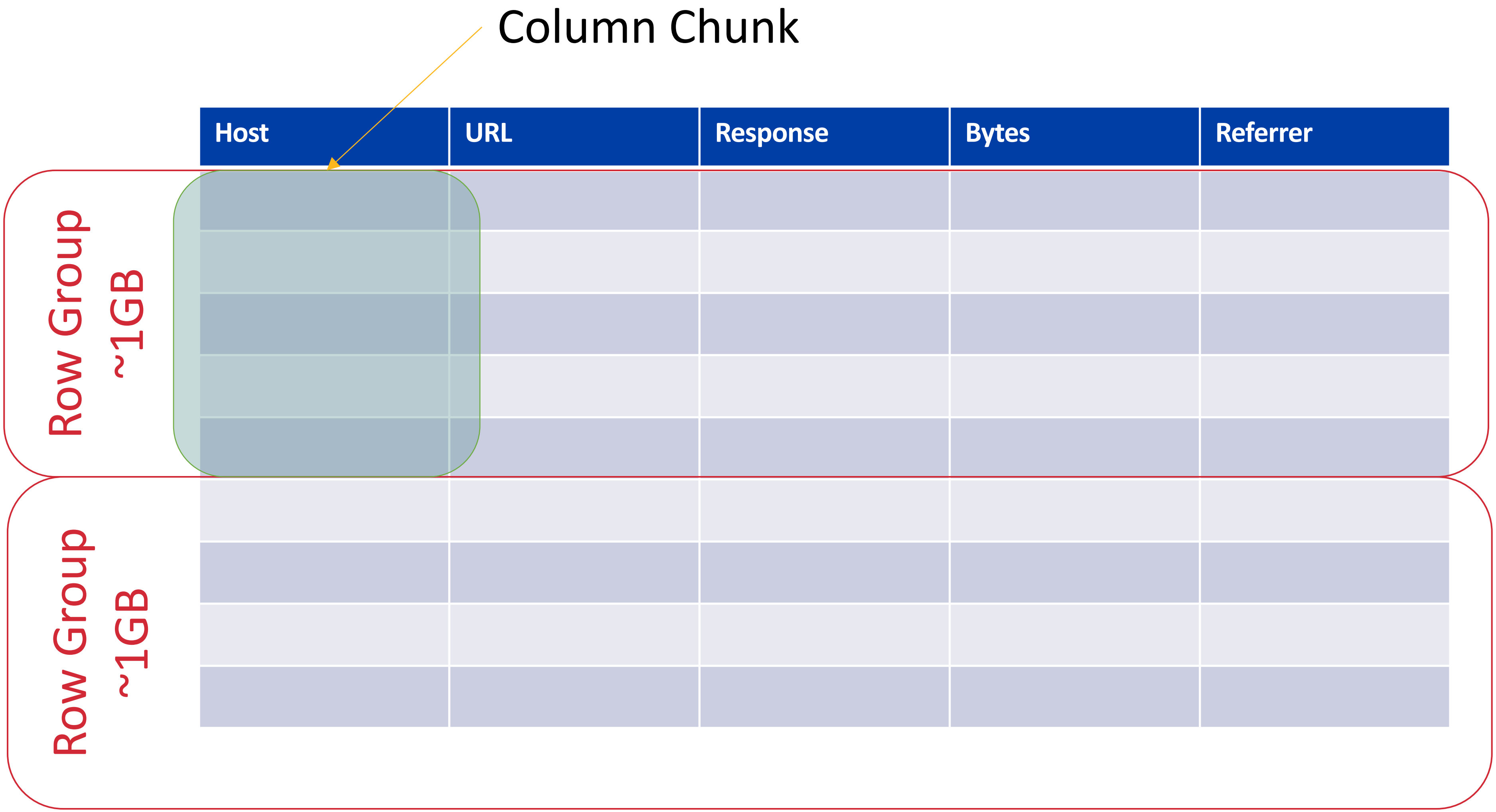
Column Format for Big Data

- The format needs to be compatible with HDFS structure to maximize data locality
- The format needs to support nesting and repetition as in JSON data

Apache Parquet

- A column format designed for big data
- Based on Google Dremel
- Designed for the distributed file system
- Supports nesting
- Language independent, can be processed in C++, Java, or other formats
- Limited to static data and recommended for analytical queries

Parquet Overview



Column Chunk

Host | URL | Response | Bytes | Referrer

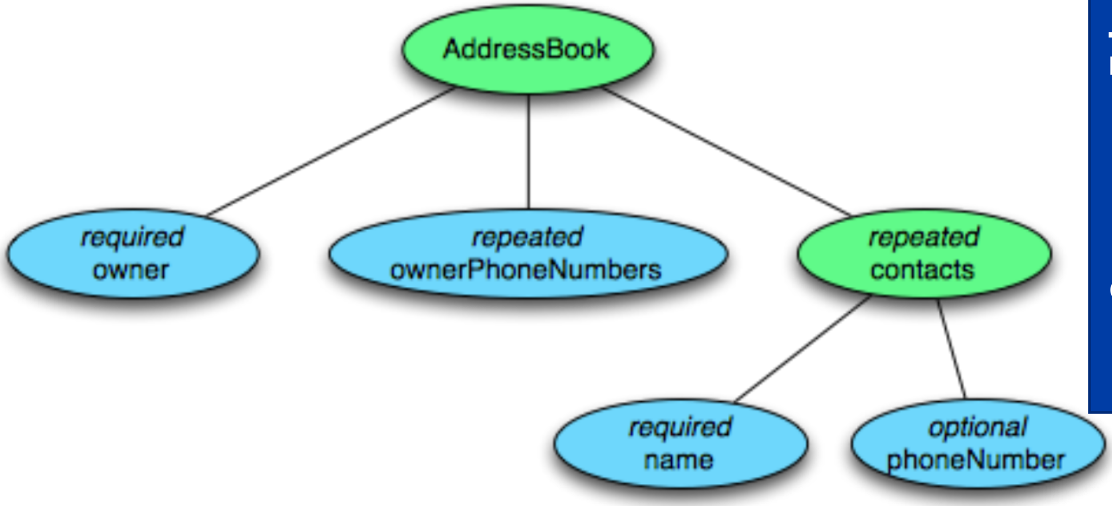
Row Group ~1GB

Row Group ~1GB

Column Chunk

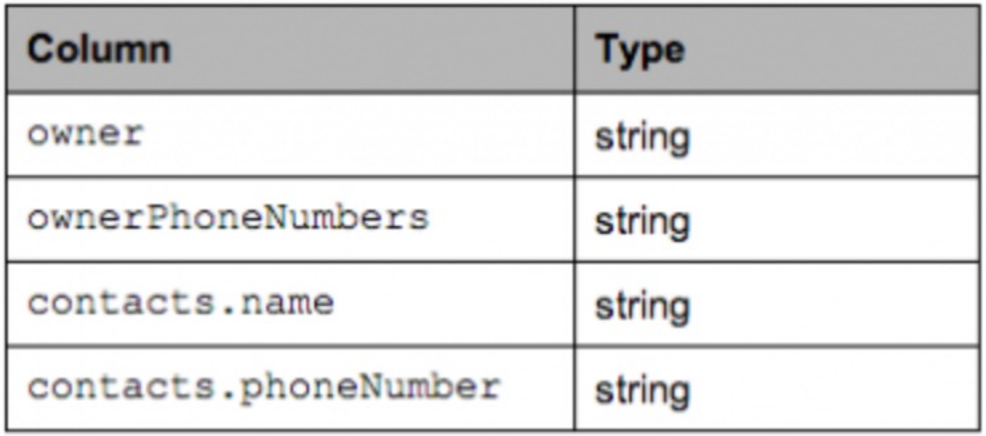
- A sequence of values of the same type
- In the absence of repetition and nesting, storing one column chunk is straight-forward
- We can store all values as a list
- Values can be compressed or encoded using any of the popular method
- When compressed, each column chunk is further split into *pages* of 16KB each
- Nesting, Repetition, and Nulls, Oh My!

Nesting and Null in Parquet

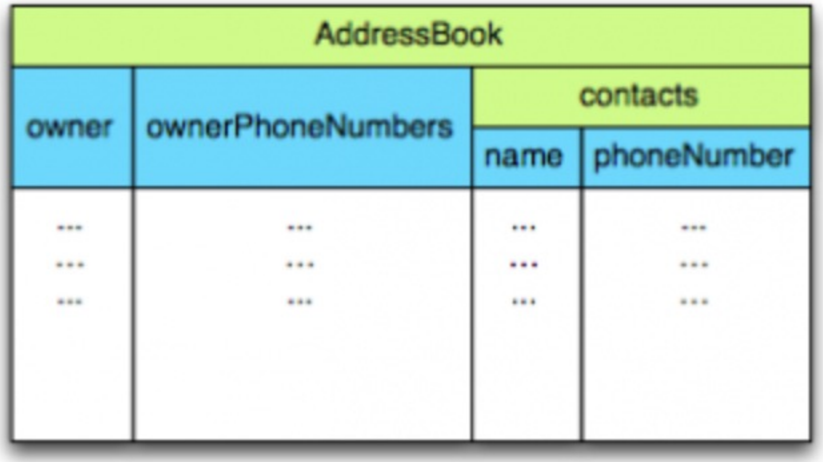


Record Schema

```
message AddressBook {
required string owner;
repeated string ownerPhoneNumbers;
repeated group contacts {
required string name;
optional string phoneNumber;
}
}
```



Column	Type
owner	string
ownerPhoneNumbers	string
contacts.name	string
contacts.phoneNumber	string



AddressBook			
owner	ownerPhoneNumbers	contacts	
		name	phoneNumber
...	...	...	...
...	...	...	...
...	...	...	...

Examples

```
message1: {
 owner: “Alex”;
 ownerPhoneNumbers: [
  “951-555-7777”, “961-555-9999”
 ],
 contacts: [{
   name: “Chris”;
   phoneNumber: “951-555-6666”;
 }]
}
```

```
message2: {
 owner: null;
 ownerPhoneNumbers: [
  “951-555-7777”, “961-555-9999”
 ],
 contacts: [{
   name: “Chris”;
   phoneNumber: “951-555-6666”;
 }]
}
```

```
message3: {
 owner: “Joe”;
 ownerPhoneNumbers: [
  “951-555-4444”, “961-555-3333”
 ]
}
```

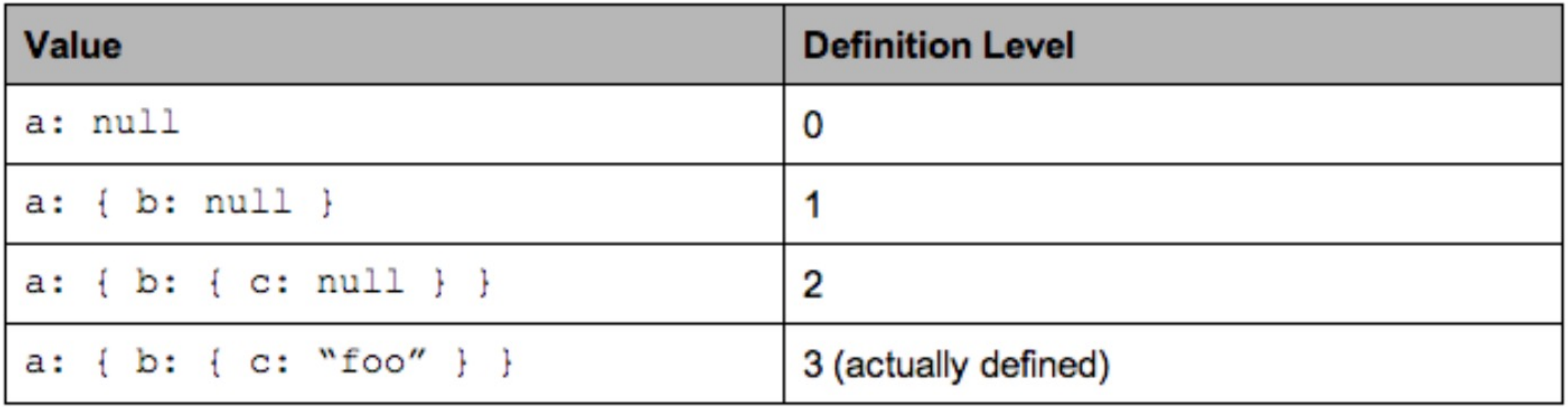
```
message4: {
 owner: “Olivia”;
 ownerPhoneNumbers: [
  “951-555-2222”
 ],
 contacts: [{
   name: “Chris”;
   phoneNumber: null;
 }]
}
```

```
message5: {
 owner: “Violet”;
 ownerPhoneNumbers: [
  “961-555-1111”
 ]
}
```

Definition Level

- The nesting level at which a field is null

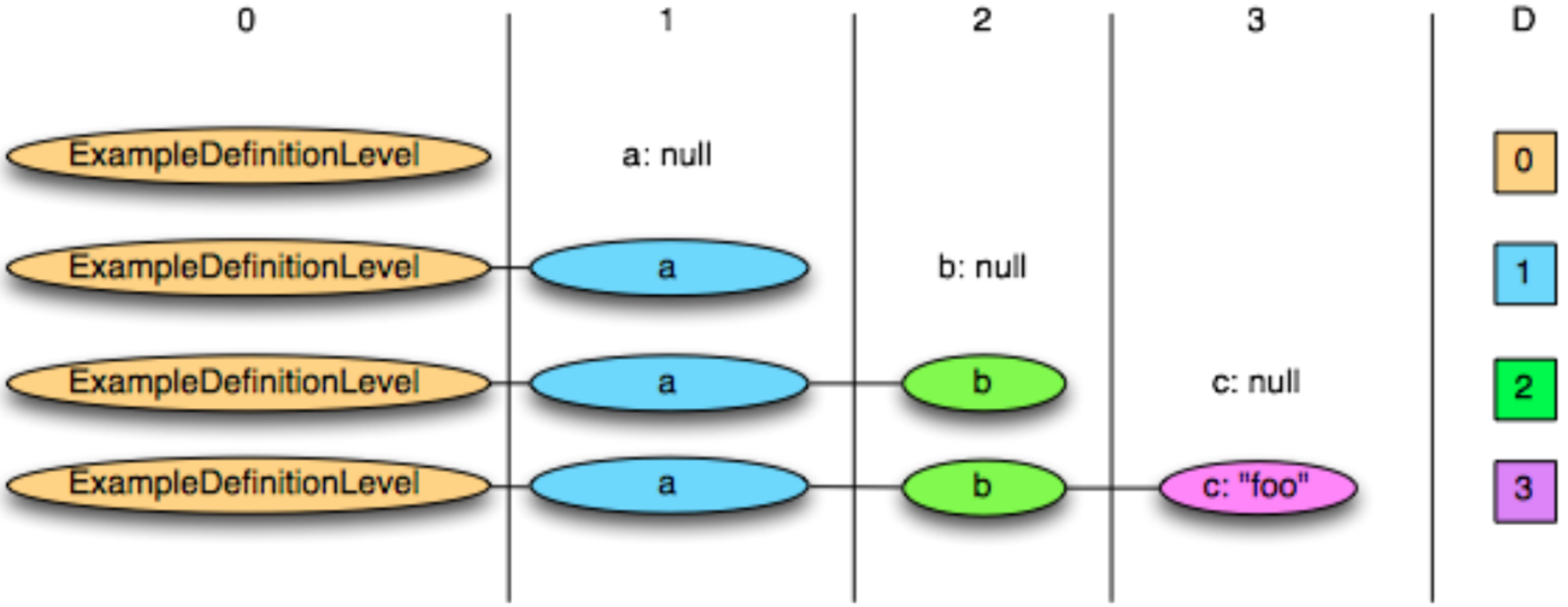
```
message ExampleDefinitionLevel {
optional group a {
optional group b {
optional string c;
}
}
}
```



Value	Definition Level
`a: null`	0
`a: { b: null }`	1
`a: { b: { c: null } }`	2
`a: { b: { c: “foo” } }`	3 (actually defined)

Definition Level

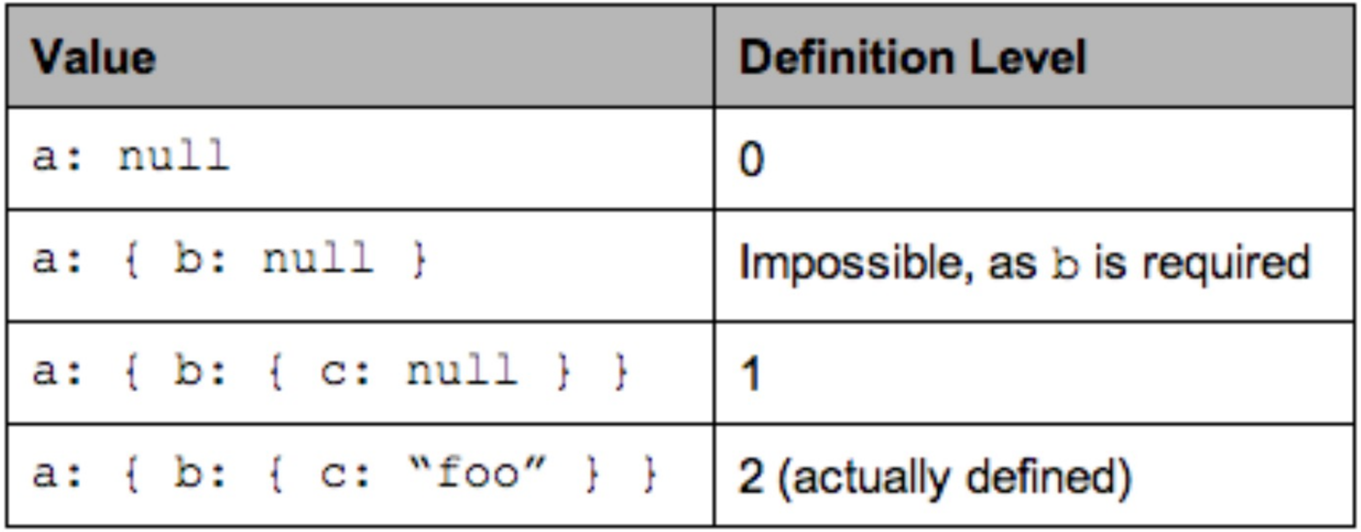
Value	Definition Level
`a: null`	0
`a: { b: null }`	1
`a: { b: { c: null } }`	2
`a: { b: { c: “foo” } }`	3 (actually defined)



Definition Level with Required

- When a field is require (not nullable), then there is one definition level that is not allowed

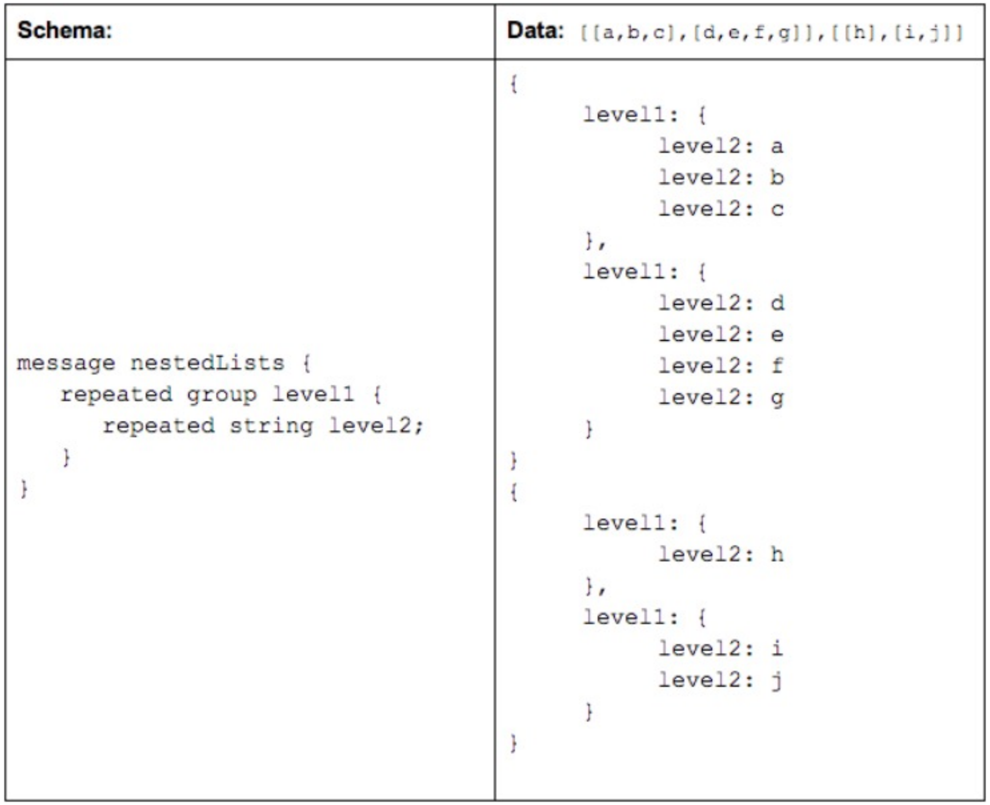
```
message ExampleDefinitionLevel {
optional group a {
required group b {
optional string c;
}
}
}
```



Value	Definition Level
`a: null`	0
`a: { b: null }`	Impossible, as `b` is required
`a: { b: { c: null } }`	1
`a: { b: { c: "foo" } }`	2 (actually defined)

Repetition Level

- The level at which we should create a new list

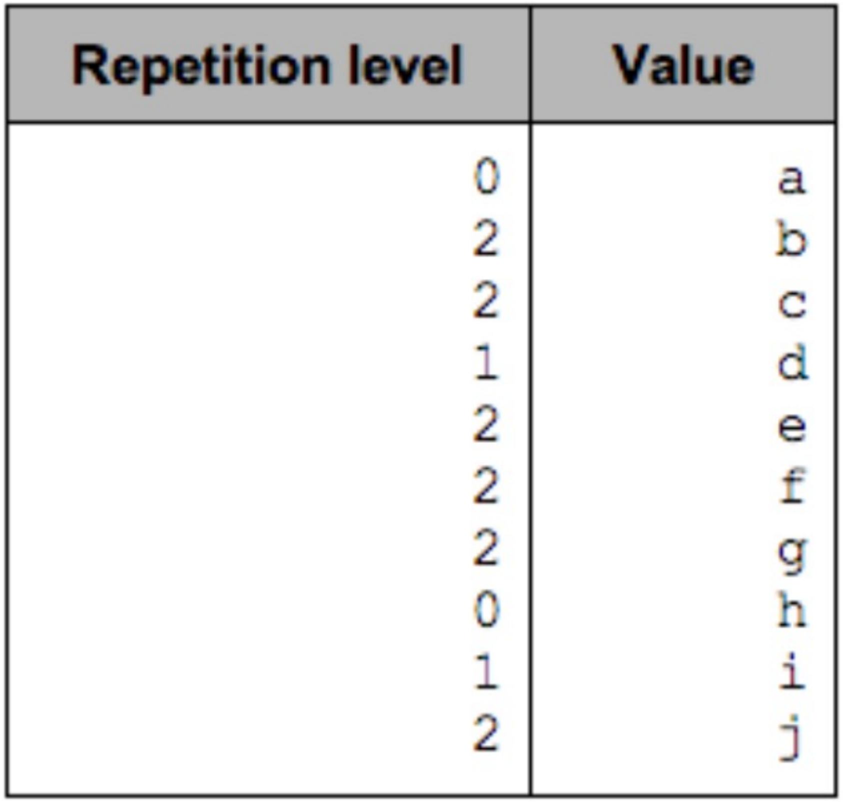


Schema:

```
message nestedLists {
    repeated group level1 {
        repeated string level2;
    }
}
```

Data: [[a,b,c],[d,e,f,g]],[[h],[i,j]]

```
{
        level1: {
                level2: a
                level2: b
                level2: c
        },
        level1: {
                level2: d
                level2: e
                level2: f
                level2: g
        }
}
{
        level1: {
                level2: h
        },
        level1: {
                level2: i
                level2: j
        }
}
```

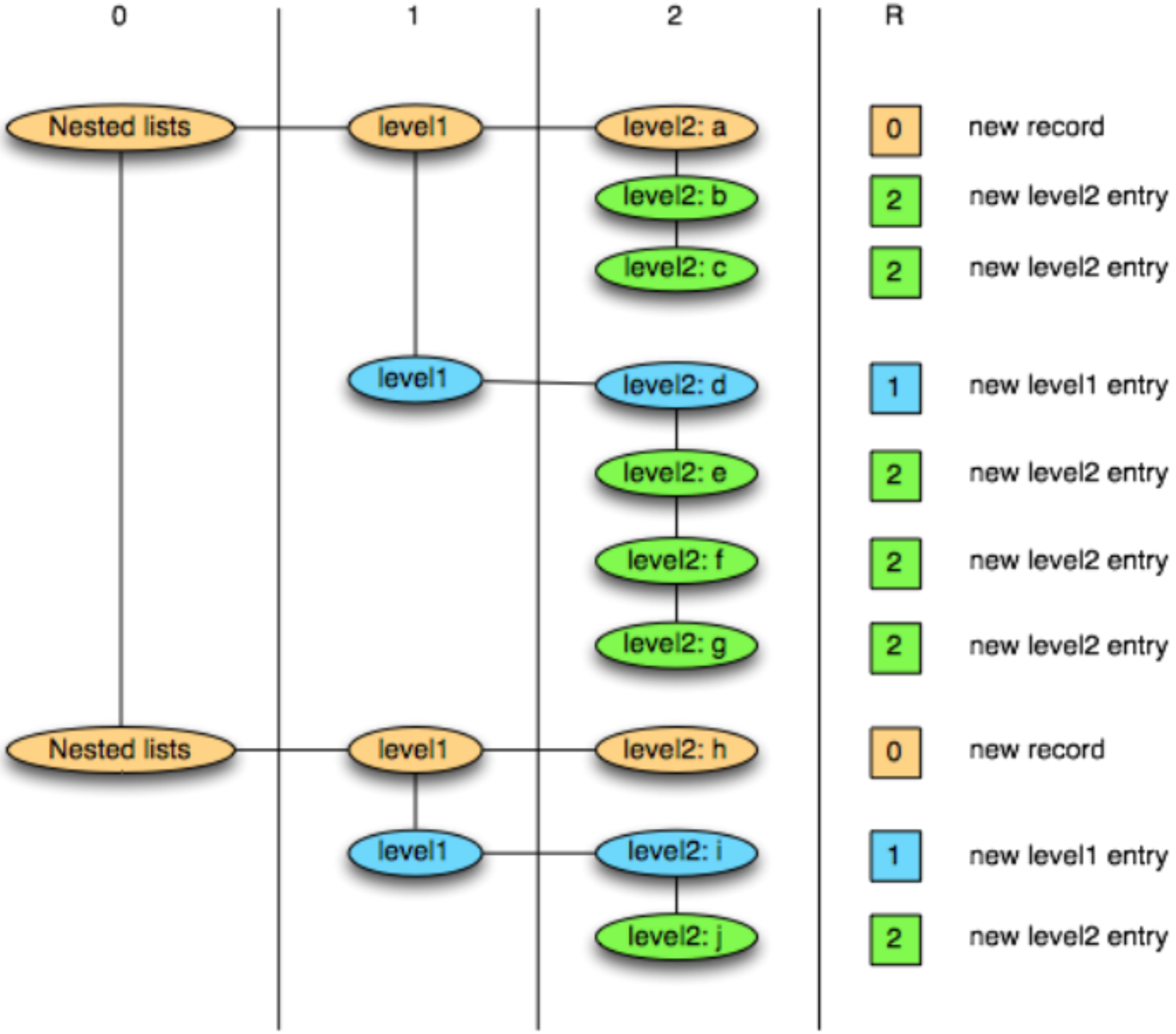


Repetition level	Value
0	a
2	b
2	c
1	d
2	e
2	f
2	g
0	h
1	i
2	j

Repetition Level

- The repetition level marks the beginning of lists and can be interpreted as follows:
  - 0 marks every new record and implies creating a new level1 and level2 list
  - 1 marks every new level1 list and implies creating a new level2 list as well.
  - 2 marks every new element in a level2 list.

Repetition Level



AddressBook Example

Record Schema

```
message AddressBook {
required string owner;
repeated string ownerPhoneNumbers;
repeated group contacts {
required string name;
optional string phoneNumber;
}
}
```

Attribute	Optional	Max Definition level	Max Repetition level
Owner	No	0 (owner is required)	0 (no repetition)
Owner phone number	Yes	1	1 (repeated)
Contacts.name	No	1 (name is required)	1 (contacts is repeated)
Contacts.Phone number	Yes	2 (phone is optional)	1 (contacts is repeated)

Example

```
DocId: 10
Links
    Forward: 20
    Forward: 40
    Forward: 60
Name
    Language
        Code: ‘en-us’
        Country: ‘us’
    Language
        Code: ‘en’
    Url: ‘http://A’
Name
    Url: ‘http://b’
Name
    Language
        Code: ‘en-gb’
        Country: ‘gb’
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated in64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    option String Url;}}
```

```
DocId: 20
Links
    Backward: 10
    Backward: 30
    Forward: 80
Name
    Url: ‘http://C’
```

Further Reading

- **Dremel made simple with Parquet** **[**https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html]
- Apache Parquet project homepage [http://parquet.apache.org]
- Parquet for MapReduce (works for both Hadoop and Spark) [https://github.com/apache/parquet-mr]